

FAQ

Q How does a dynamic language, such as Ruby, differ from languages such as C# or Java?

A In conventional languages, such as C# or Java, there is a division between writing the program and running it that is less clear for a dynamic language. When you program in a dynamic language, commands you type are obeyed at once. If you type 'draw a square', a square appears on the screen. This instant feedback tells you immediately if there is a problem with your command, so you can correct it at once.

Dynamic languages can be very productive. They are not as efficient as conventional languages, but they are object-oriented and have the potential to be as structured. You can try a dynamic language for free by downloading Ruby (www.ruby-lang.org) or Python (www.python.org).

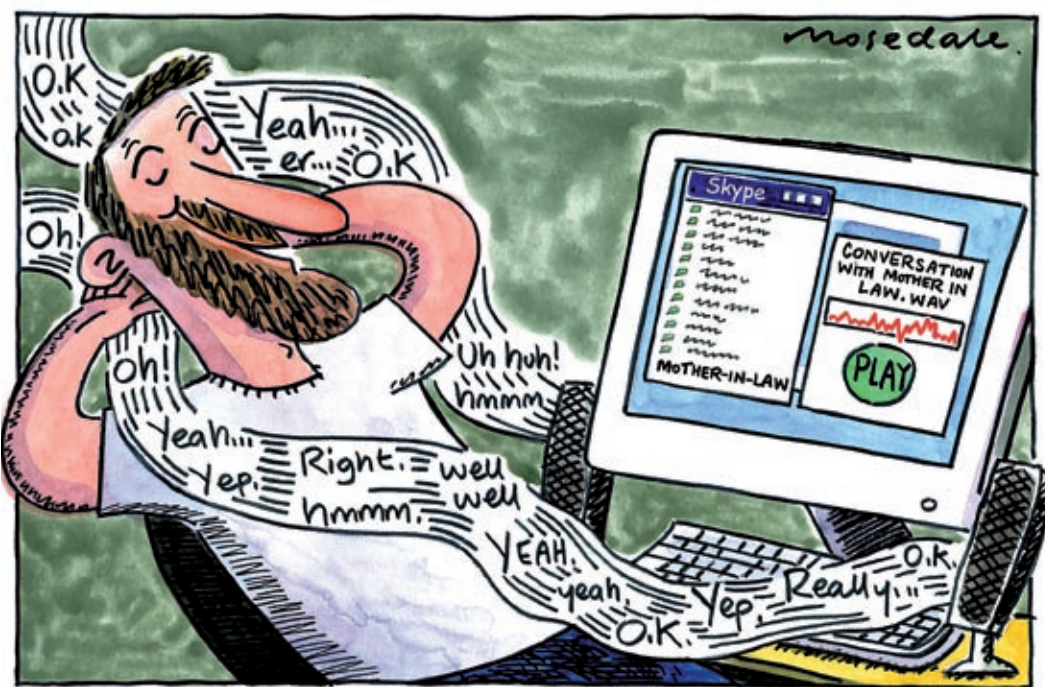
Mike James

IT author, lecturer
and programmer

mike@computershopper.co.uk

Recording Skype

Want to write your own Skype voicemail, or be able to play an audio file as part of a conversation? Mike James shows you how to use Skype's API to do this and more



Skype is a great and easy way of using VoIP to make cheap phone calls over the internet. It's also a really interesting application from a programmer's point of view, as its API is very easy to use. Skype's API is available both as an ActiveX control and a raw API. We looked at the ActiveX control in *Shopper* November 2006. It's incredibly useful but rather limited in its scope, as it doesn't provide full access to audio. Fortunately, the full API does, and mastering it enables you to develop lots of exciting projects, such as writing your own answering machine, or being able to play the contents of an audio file to the person on the other end of a call.

While the API's documentation and the available examples may give you the impression that it's complicated and difficult to learn, it's not actually that bad. Despite there being very little information on how to use the Skype API from a managed language such as C#, it's all just a matter of message passing – a topic that we covered in *Shopper* July 2007.

A key fact about Windows that sophisticated programming systems such as .NET manage to hide is that it is a message-passing operating system. All those controls, buttons and text boxes are just specialised windows that communicate with one another by sending messages. When you click a button, for example, it generates a message and passes it on to its parent window. This window then performs some action in response to the click. Languages such as C# use an alternative concept of events to do the same sort of thing, but events are built on top of messages. The good news is that .NET provides easy access to the message-processing machinery of an application, and you can use the Windows API to send messages to other applications.

So, rather than using COM or some other more sophisticated mechanism, the Skype API works by

passing custom messages. To control Skype, the application you develop has to send particular messages that contain commands in text form. Skype then sends messages back to your application to indicate its current status and the progress of any action you have asked it to perform.

CONNECTING TO SKYPE

Your application should check the Registry to see if Skype is installed, and present a message to the user if it isn't. For the sake of simplicity, though, we'll assume that Skype is installed, running and working properly. You should make a test call first before proceeding.

The first task is to make a connection between Skype and our application. To do this we must use two custom messages: one to make first contact, and the other to make the connection. First we have to ask Windows to give us the message code that corresponds with the custom message, which we do by using the `RegisterWindowMessage` API call. If the message isn't already registered, Windows assigns a new unique code. If another application has already registered the message, it returns the code previously allocated.

Start a new C# Windows application using C# Express or Visual Studio – all the ideas here also work in Visual Basic or any other .NET language with minor modifications. The first thing we need is the definition of the API call:

```
[DllImport("user32.dll")]  
static extern uint RegisterWindowMessage(  
    string lpString);
```

As we are using interop services to call the Windows API, we also need to add:

```
using System.Runtime.InteropServices;
```

to the start of the program. Next, we need two global variables to store the codes of the two custom messages:

```
public UInt32 APIDiscover;
public UInt32 APIAttach;
```

Now we can call the API twice to find out the codes. Start a new method called `SkypeFirstConnect`. The purpose of this, as its name suggests, is to make the very first connection to Skype. You may also want to write a reconnect method to use in the event that Skype goes offline temporarily.

```
private void SkypeFirstConnect()
{
    APIDiscover =
        RegisterWindowMessage(
            "SkypeControlAPIDiscover");
    APIAttach = RegisterWindowMessage(
        "SkypeControlAPIAttach");
}
```

As long as Skype is installed on the system, we should have the two custom message codes stored in the appropriate variables. We can now send the `APIDiscover` message to every window in the system and wait for Skype to send the `APIAttach` message back to us. In general, working with the Skype API is a matter of sending a message and then getting a response. We need a general method of waiting for the response, which sometimes has some associated text data. The simplest thing to do is to define a struct:

```
public struct databuffer
{
    public string data;
    public Boolean valid;
}
```

This has a field for holding the returned data and a Boolean field to indicate when the data is valid – in other words, when a message has been sent back from Skype. We need a single global instance of this struct:

```
public databuffer skypedata;
```

Continuing the `SkypeFirstConnect` method, we must first invalidate the return data and use the API call `SendMessageTimeout` to send the `APIDiscover` message to all active windows:

```
skypedata.valid = false;
skypedata.data = string.Empty;
IntPtr result;
IntPtr aRetVal = SendMessageTimeout(
    HWND_BROADCAST,
    APIDiscover,
    this.Handle,
    IntPtr.Zero,
    SendMessageTimeoutFlags.SMT_
        NORMAL,
    100,
    out result);
```

This API call waits for the message to be processed, or for a maximum length of time that you specify – 100 milliseconds in this case. The first parameter sends the `APIDiscover` message to all windows, the third parameter passes our application's window handle and the final few parameters control how the API call behaves.

Check the documentation for the fine detail, but the following enumeration helps:

```
[Flags]
public enum SendMessageTimeoutFlags
: uint
{
    SMT_NORMAL = 0x0000,
    SMT_BLOCK = 0x0001,
    SMT_ABORTIFHUNG = 0x0002,
    SMT_NOTIMEOUTIFNOTHUNG = 0x0008
}
```

To make this work we also need a definition of the API call and a value for `HWND_BROADCAST`:

```
[DllImport("user32.dll")]
public static extern IntPtr
SendMessageTimeout(
    IntPtr windowHandle,
    uint Msg,
    IntPtr wParam,
    IntPtr lParam,
    SendMessageTimeoutFlags flags,
    uint timeout,
    out IntPtr result);

public static readonly IntPtr
    HWND_BROADCAST = new IntPtr(-1);
```

The method has now completed its task, and we simply wait for Skype to send a message back to us. Exactly how this works is explained next, but for the moment we must decide whether to make the method 'blocking' or 'asynchronous'. We could wait within the method until Skype responds, which would block any further processing by the application, or we could return and rely on some other part of the application to handle the response asynchronously. In most cases asynchronous operation is the more powerful, and if you plan to use the Skype API further, you should implement an asynchronous approach by generating an event when the response arrives.

However, blocking is simpler, and using it makes the way everything works more obvious. So let's implement a `WaitForSkype` method that waits for Skype to send back a response containing a specified keyword, and simply use it at the end of the method:

```
WaitForSkype("");
}
```

If the target keyword is a null string, the method simply waits for any response from Skype. The code for the `WaitForSkype` function is fairly simple – just two nested loops, the outer one waiting for valid data from Skype and the inner one checking for the keyword:

```
private void WaitForSkype(string
    target)
{
    do
    {
        Application.DoEvents();
    } while (!skypedata.valid);
    while (!skypedata.data.
        Contains(target));
}
```

MESSAGE IN A QUEUE

At this point, you may be wondering how the `skypedata` struct ever gets any data in it. At the moment we haven't touched on this second half of the interchange. So far our application sends a message to Skype but ignores any response. How do we pick up a message that another application has sent?

The answer is that we have to tap into the message loop, or 'pump', that every application has. In many cases the details of message processing are hidden deep within the language or framework, but .NET provides easy access. Every form has a virtual `WndProc` method that implements the message loop. To process additional messages, all you have to do is override it. In this case we simply add:

```
protected override void WndProc(ref
    Message m)
{
}
```

`WndProc` is called each time there is a message to process. The parameter `m` is a struct that contains the different portions of the message. The internal workings of Windows don't use a struct or any sort of data structure to represent a message; this is just .NET wrapping the lower-level system functions for us.

The type of message is determined by the `Msg` property, and we know from the documentation that Skype will return the custom `APIAttach` message that we retrieved the code for earlier. To process this we simply use an `if` statement to detect the message:

```
if ((UInt32)m.Msg==APIAttach)
{
}
```

Skype uses the `LParam` parameter in the message to send a status code along with the message. The easiest way to decode this is to declare an enumeration:

```
public enum SkypeAttachStatus : uint
{
    Success = 0,
    PendingAuthorization = 1,
    Refused = 2,
    NotAvailable = 3,
    Available = 0x8001
}
```

If you want to see the status, place a `Textbox` on the form and add:

```
SkypeAttachStatus Status =
    (SkypeAttachStatus)m.LParam;
textBox1.Text += "Status=" +
    Status.ToString() +
    Environment.NewLine;
```

When you run the finished program, you will see that Skype sends a stream of messages in response to the `APIDiscover` message you sent, each of which has a different status. You can write code that reacts to each of the different status conditions, but the only one that really matters is `Success`, which indicates that you have permission to use Skype further. In this case `WParam` is returned containing a handle to Skype's main window, which you use to send further messages to it.

We clearly need to store Skype's window handle for further use, so we need an additional

VISUAL BLUEPRINT Visual Basic 2005

★★★

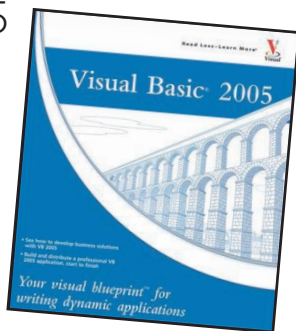
£13

This is another book in the Visual Blueprint series that takes a fully illustrated, step-by-step approach to its topics. The author, Jim Keogh, has taken the standard format and applied it to explaining Visual Basic, making very few concessions to the subject matter in the process.

Visual Basic 2005 uses plenty of pictures and touches on almost every topic on the subject. The main problem is that there's never enough detail, and many of the pictures aren't particularly helpful. The book covers the basics, including adding a button, adding a text box and adding a combo box, but makes no real effort to point out or use the general principles.

The same is true of the rest of the book, which is light on principles and heavy on step-by-step guides. As each instalment is very short, you never get to see an entire program or anything that amounts to a project.

The main problem with *Visual Basic 2005* is that it provides short, repetitive introductions to very specific aspects of Visual Basic. This isn't a good way to learn a programming language, and nor does it serve those who are already familiar with Visual Basic. There are better books to learn from, and *Visual Basic 2005* is restricted by having to stick to Visual Blueprint's fully illustrated layout.



SUMMARY

- ✓ Thorough, with lots of pictures
- ✗ Content determined by format

SUPPLIER www.amazon.co.uk
ISBN 0471793442
PAGES 320

SITEPOINT Simply JavaScript

★★★★★

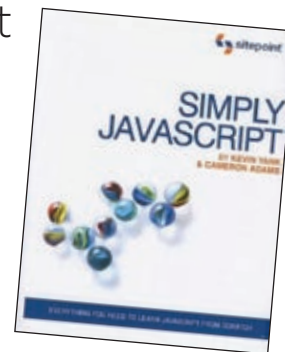
£27

JavaScript has been around for a while, but with Ajax and the emphasis on object-oriented JavaScript, it's arguably more important than ever. If you're looking to develop your skills, *Simply JavaScript* is a good starting point for anyone with a bit of experience.

Kevin Yank is clearly an experienced JavaScript programmer, which shows with the level of copy, and there are lots of things that will interest intermediate and expert readers. It neatly emphasises the complexities of having to deal with the reality of browser incompatibilities, and we learned a lot from these parts of the book.

Yank's approach to HTML is a little disorganised and he tends to depend on JavaScript libraries to make the language more powerful. These are small problems, though, and *Simply JavaScript* remains a great book if you're looking to learn modern object-oriented JavaScript and Ajax.

Ultimately, there is so much more to JavaScript than this book can cover, but it does a good job of introducing the language. There are plenty of practical examples, as well as free code that you can download and practise with. Complete beginners will struggle to follow the book, but for everyone else, *Simply JavaScript* is a great introduction to an important web technology.



SUMMARY

- ✓ Deals with real-world complexities
- ✗ Disorganised approach to HTML

SUPPLIER www.amazon.co.uk
ISBN 0980285801
PAGES 424

global variable:

```
private IntPtr HSkype = IntPtr.Zero;␣
```

Now we can test the status and store the window handle for use later:

```
if (Status == SkypeAttachStatus.
    Success)␣
{␣
    HSkype = m.WParam;␣
}␣
```

However, this isn't quite the end of the story. Skype sends a final message after the Success message to say that it is available for use, and it's this one that, in a simple-minded blocking transaction, we should wait for:

```
if (Status == SkypeAttachStatus.
    Available)␣
{␣
    skypedata.valid = true;␣
}␣
```

By setting the data buffer's valid property to true we free up WaitForSkype to move on to the next operation. Next, we have to set a non-zero result to signify that the message has been processed, because without this Skype will close the connection:

```
m.Result = new IntPtr(1);␣
return;␣
}␣
```

Finally, we have to call the parent WndProc method if we haven't handled the message so that it can deal with all the messages relating to

button clicks and general user interaction. The rule is that when you override WndProc, you handle the messages you want to handle but always call the parent WndProc to process any messages you don't handle:

```
base.WndProc(ref m);␣
}␣
```

TALKING TO SKYPE

If you add a call to SkypeFirstConnect to the button's click handler and run the program, you should see Skype status messages appear in the Textbox until the 'Available' status is reached and the connection made. Now that we have Skype's attention, it's time to send it some commands. Fortunately, this is mostly more of the same. We send Skype a message and Skype sends us messages back to let us know what's happening. The new element is that the messages now contain commands and status information in the form of strings. For this we need a slightly different version of the SendMessageTimeout API call. The only difference is that the lParam parameter is used as a pointer to a CopyDataStruct:

```
[DllImport("user32.dll")]␣
public static extern IntPtr
SendMessageTimeout(␣
    IntPtr windowHandle,␣
    uint Msg,␣
    IntPtr wParam,␣
    ref CopyDataStruct lParam,␣
    SendMessageTimeoutFlags flags,␣
    uint timeout,␣
    out IntPtr result);␣
```

Of course, we also need a definition of

CopyDataStruct, which can be found in the API documentation and translated to C#:

```
[StructLayout(LayoutKind.
    Sequential)]␣
public struct CopyDataStruct␣
{␣
    public string ID;␣
    public int Length;␣
    public string Data;␣
}␣
```

This struct passes the command and data string in the Data property as part of the content of a WM_COPYDATA message. WM_COPYDATA isn't a custom message – it's a perfectly standard Windows message, and any application can use it to transfer large amounts of data. The WM_COPYDATA message has to be defined as:

```
public static readonly uint␣
WM_COPYDATA = 0x004a;␣
```

Now we have everything we need to send Skype a command. In the documentation, all the commands take the form of text with parameters. To dial a number, for example, you use the Call command followed by the number you want to call. To send a command to Skype, it's a good idea to implement a suitable method:

```
public IntPtr SendCommand(string
    Command)␣
{␣
    skypedata.valid = false;␣
    skypedata.data = string.Empty;␣
```

The Skypedata buffer has been invalidated to get ready for Skype to respond to the command. To

send the command we have to initialise a suitable CopyDataStruct:

```
CopyDataStruct data = new
CopyDataStruct();
data.ID = "1";
data.Data = Command;
data.Length = data.Data.Length + 1;
```

You can use the ID property to identify multiple commands of the same type but, as we are using blocking, one command at a time is the rule. Now we can send the command:

```
IntPtr result;
IntPtr RetVal = SendMessageTimeout(
H Skype,
WM_COPYDATA,
this.Handle,
ref data,
SendMessageTimeoutFlags.SMT_
NORMAL,
100,
out result);
return RetVal;
```

As soon as Skype gets the message it sends a WM_COPYDATA message back with the Data property of the CopyDataStruct full of text that gives the status and other information about what is happening. We simply add another if statement that checks to see if the message is a WM_COPYDATA message:

```
if ((UInt32)m.Msg == WM_COPYDATA)
{
```

We should make sure that the WM_COPYDATA message was sent by the Skype window. After all, other application windows can send WM_COPYDATA messages for a range of reasons:

```
if (m.WParam == H Skype)
{
```

As long as this is indeed a WM_COPYDATA message from Skype, we can unpack the data it contains by casting the object returned by the GetLPParam method:

```
CopyDataStruct data =
(CopyDataStruct)m.GetLPParam(
typeof(CopyDataStruct));
```

The GetLPParam method uses the LParam value to point to an area of memory, which it then converts to an object of the type you specify.

Finally, we move the returned data into the skypedata buffer, display it in the textbox and set the buffer valid indicator to true:

```
skypedata.data = data.Data;
m.Result = new IntPtr(1);
textBox1.Text += "Skype data=" +
skypedata.data +
Environment.NewLine;
skypedata.valid = true;
return;
```

MAKE THE CALL

That's all there is to it. We now have complete mechanisms for sending a message and getting

the response. The easiest way to see them in action is to use our application to call the Skype testing service. Enter the following code in the button's click handler:

```
private void button1_Click(object
sender,
EventArgs e)
{
SkypeFirstConnect();
IntPtr result = SendCommand("CALL
echo123");
WaitForSkype("INPROGRESS");
}
```

If you run the program and click the button, you will see the various Skype status messages scrolling past. The WaitForSkype allows the program to continue as soon as the call is connected. Of course, you need to write some code to handle the possibility that the call will fail, but this at least demonstrates the overall principle of how things are done.

Now that we have implemented the basics, finding out what else we can do is mostly a matter of learning what commands Skype supports and how to use them. Sometimes you will need to add additional processing to the existing methods. For an example of the sort of powerful things you can easily do, let's record the incoming side of a call to a file. The command we need for the job is 'ALTER CALL called SET OUTPUT destination'. The problem is that to use it we need the call's ID, which is returned as part of the status message during the call. Extracting the ID from the status is fairly easy, and we can create a method to do it:

```
private string getCallId()
{
string[] items = skypedata.data.
Split(' ');
return items[1];
}
```

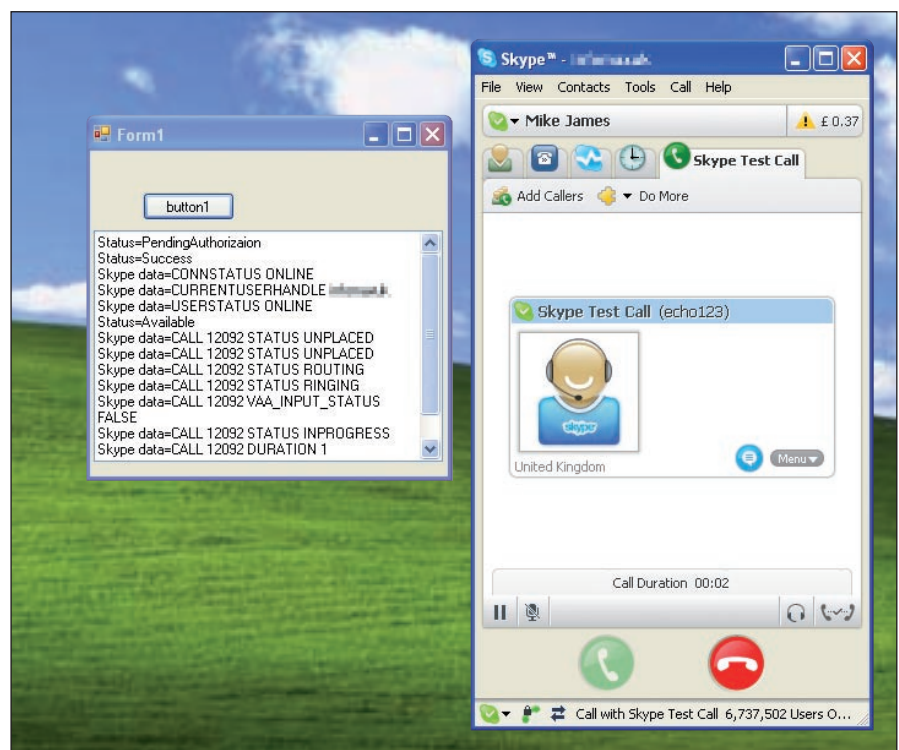
This simply splits the string into different words separated by spaces and returns the second item – remember the first item is items[0] which is always the call ID in a call status message. Now we simply have to redefine the button's click event handler:

```
private void button1_Click(object
sender,
EventArgs e)
{
SkypeFirstConnect();
IntPtr result = SendCommand("CALL
echo123");
WaitForSkype("INPROGRESS");

string CallID = getCallId();
string filename = @"c:\test.wav";
string cmd = "ALTER CALL" + CallID
+
"SET_OUTPUT file=" + filename;
result = SendCommand(cmd);
WaitForSkype("FINISHED");
}
```

Once we know that the call is in progress, we get the call ID, construct the text for the command to save the audio output in a file called test.wav and send the command. When the call is FINISHED you can open the test.wav file and hear the Skype testing service go through its usual script. You can divert the input audio stream in exactly the same way to build your own answering machine or automatic messaging system.

If you want to do something like this, don't simply use the example program listed here. You need to implement a Skype object complete with asynchronous methods and events, and you need to put in lots of error checking and recovery. It's not difficult, but it would have made our example code much harder to understand. ☞



▲ You can see the status messages scrolling past as the call progresses